Concurrent and Reactive Constraint Programming

Maurizio Gabbrielli, Catuscia Palamidessi, Frank D. Valencia

Concurrent and Reactive Constraint Programming

Maurizio Gabbrielli
Lab. Focus, INRIA and University of Bologna

Catuscia Palamidessi
INRIA and LIX, Ecole Polytechnique

Frank D. Valencia
CNRS, LIX, Ecole Polytechnique

Abstract. The Italian Logic Programming community has given several contributions to the theory of Concurrent Constraint Programming. In particular, in the topics of semantics, verification, and timed extensions. In this paper we review the main lines of research and contributions of the community in this field.

1 The origins: from concurrect logic programming to concurrent constraint programming

In the 80's there had been several proposals to extend logic programming with constructs for concurrency, aiming at the development of a concurrent language which would maintain the typical advantages of logic programming: declarative reading, computations as proofs, amenability to meta-programming etc. Examples of concurrent logic languages include PARLOG [14], Concurrent Prolog [58, 59], Guarded Horn Clauses (GHC) [61, 62] and their so-called *flat* versions. Towards the end of the decade, Concurrent constraint programming ([52, 56, 57]) emerged as one of the most successful proposals in this area.

Concurrent constraint programming (ccp) presented two new perspectives on the underlying philosophy of logic programming. One is the replacement of the concept of unification over the Herbrand universe by the more general notion of *constraint* over an arbitrary domain. This is in a sense a 'natural' development, and the idea was already introduced in 'sequential' logic programming by Jaffar and Lassez ([45]). The other is the introduction of extra-logical *operators* typical of the imperative concurrent paradigms, like CCS ([47]), TCSP ([8]) and ACP ([1]); in particular, the *choice* ($+$), the *action prefixing* ($\rightarrow$), and the *hiding* operator ($\exists$). Additionally, concurrent constraint programming embodies an explicit characterization of the *control* mechanisms for communication and synchronization by means of the introduction of two kinds of actions (*ask* and *tell*). Also in concurrent logic languages these control features were present, but they were hidden in various ways: the choice was represented by alternative clauses, hiding by local (existentially quantified) variables, prefixing by commitment, communication by sharing of variables, and synchronization by restrictions on the unification algorithm.

There are many advantages in an explicit representation of these concurrency control mechanisms by means of operators. First of all, they are 'isolated' and therefore the laws of their behaviour can be understood better. For instance, one of the problems in studying the semantics of concurrent logic programming is that the choice mechanism is 'mixed up' with recursion, since a clause is in general a recursive definition. Second, the standard tools developed in the theory of concurrency can be applied more easily. Third, a 'reconciliation' with the declarative principles of logic programming is more feasible, once the basic limitations are well understood. For instance, the conditions which rule the behaviour of *ask* and *tell* can be described in a logical way, thus providing the synchronization mechanism with a 'declarative flavour' ([46, 51]) that was missing in the 'restricted-unification' approach.

2 The ccp paradigm

Ccp is based on the concept of *store-as-constraint*, in contrast to von Neumann's concept of *store-as-valuation*. The computation proceeds through the concurrent execution of different processes, which interact and communicate through the common store. They refine the partial information about the values of the variables by adding (*telling*) constraints to the store, and they test (*ask*) whether the store entails a constraint before proceeding in the computation.

One of the most characteristic features of the ccp paradigm is a formalization of these basic operations which allow to update and to query the common store, in terms of the logical notions of consistency, conjunction and entailment supported by a given underlying constraint system.

Here we recall briefly the syntax and semantics of ccp. Among the several variants which have been proposed in literature, we choose the simplest and most basic one, called *eventual tell* ccp. Most of the other ccp dialects can be obtained by enriching this one.

The ccp languages are defined parametrically w.r.t. to a given *cylindric constraint system*.

Definition 1.

- *A constraint system is a complete algebraic lattice* $\langle \mathcal{C}, \vdash, \sqcup, true, false \rangle$ *where* $\sqcup$ *is the lub operation, and true, false are the least and the greatest elements of* $\mathcal{C}$*, respectively. The entaiment relation* $\vdash$ *is the inverse ordering.*
- *Consider a (denumerable) set of variables* $x, y, z, \ldots$*. Assume that for each* $x \in Var$ *a function* $\exists_x : \mathcal{C} \to \mathcal{C}$ *is defined such that for any* $c, d \in \mathcal{C}$*:*
  (i) $c \vdash \exists_x(c)$,
  (ii) if $c \vdash d$ *then* $\exists_x(c) \vdash \exists_x(d)$,
  (iii) $\exists_x(c \sqcup \exists_x(d)) = \exists_x(c) \sqcup \exists_x(d)$,
  (iv) $\exists_x(\exists_y(c)) = \exists_y(\exists_x(c))$.
  Then $\mathbf{C} = \langle \mathcal{C}, \leq, \sqcup, true, false, Var, \exists \rangle$ *is a cylindric constraint system.*

In order to model parameter passing, *diagonal elements* ([44]) are added to the primitive constraints: We assume that, for x, y ranging in Var, D contains the constraints d_{xy} which satisfy the following axioms.

(i) $true \vdash d_{xx}$,
(ii) if $z \neq x, y$ then $d_{xy} = \exists_z(d_{xz} \sqcup d_{zy})$,
(iii) if $x \neq y$ then $d_{xy} \sqcup \exists_x(c \sqcup d_{xy}) \vdash c$.

Note that if $\mathbf{C}$ models the equality theory, then the elements d_{xy} can be thought of as the formulas $x = y$. In the following $\exists_x(c)$ is denoted by $\exists_x c$ with the convention that, in case of ambiguity, the scope of $\exists_x$ is limited to the first constraint subexpression. (So, for instance, $\exists_x c \sqcup d$ stands for $\exists_x(c) \sqcup d$.)

Definition 2. *Assuming a given cylindric constraint system* $\mathbf{C}$ *the syntax of* agents *is given by the following grammar:*

$$A ::= stop \mid tell(c) \mid \sum_{i=1}^{n} ask(c_i) \rightarrow A_i \mid A \parallel A \mid \exists x A \mid p(x)$$

where the c, c_i *are supposed to be* finite constraints *(i.e. algebraic elements) in* $\mathcal{C}$*. A ccp* process P *is then an object of the form* $D.A$*, where* D *is a set of procedure declarations of the form* $p(x) :: A$ *and* A *is an agent.*

The *deterministic* agents are obtained by imposing the restriction $n = 1$ in the previous grammar. The standard operational model of ccp can be described by a transition system $T = (Conf, \longrightarrow)$. The configurations (in) $Conf$ are pairs consisting of a process, and a constraint in $\mathcal{C}$ representing the common *store*. The transition relation $\longrightarrow \subseteq Conf \times Conf$ is described by the (least relation satisfying the) rules **R1-R6** of table 1.

The agent *stop* represents successful termination. The basic actions are given by $tell(c)$ and $ask(c)$ constructs which act on the common store. Given a store d, as shown by rule **R1**, the execution of $tell(c)$ updates the store to $c \sqcup d$. The action $ask(c)$ represents a guard, i.e. a test on the current store d, whose execution does not modify d. We say that $ask(c)$ is *enabled* in d iff $d \vdash c$. According to rule **R2** the *guarded choice* operator gives rise to global non-determinism: the agent $\sum_{i=1}^{n} ask(c_i) \rightarrow A_i$ nondeterministically selects one $ask(c_i)$ which is enabled in the current store, and then behaves like A_i. The external environment can then affect the choice since $ask(c)$ is enabled iff the current store d entails c, and d can be modified by other agents (rule **R1**). If no guard is enabled, then the guarded choice agent *suspends*, waiting for other (parallel) agents to add information to the store. The situation in which all the components of a system of parallel agents suspend is called *global suspension* or *deadlock*. The operator $\parallel$ represents parallel composition which is described by rule **R3** as interleaving. The agent $\exists x A$ behaves like A, with x considered *local* to A. To describe locality in rule **R4** the syntax has been extended by an agent $\exists^d x A$ where d is a local store of A containing information on x which is hidden in the external store. Initially the local store is empty, i.e. $\exists x A = \exists^{true} x A$.

Rule **R5** treats the case of a procedure call when the actual parameter equals the formal parameter: in this case a simple body replacement suffices. We do not need more rules since, for the sake of simplicity, we assume that the set D of procedure declarations is closed w.r.t. parameter names.

R1	$\langle D.tell(c), d\rangle \longrightarrow \langle D.Stop, c \sqcup d\rangle$
R2	$\langle D. \sum_{i=1}^{n} ask(c_i) \rightarrow A_i, d\rangle \longrightarrow \langle D.A_j, d\rangle \ j \in [1,n] \ and \ d \vdash c_i$
R3	$\dfrac{\langle D.A, c\rangle \longrightarrow \langle D.A', c'\rangle}{\begin{array}{l}\langle D.A \parallel B, c\rangle \longrightarrow \langle D.A' \parallel B, c'\rangle \\ \langle D.B \parallel A, c\rangle \longrightarrow \langle D.B \parallel A', c'\rangle\end{array}}$
R4	$\dfrac{\langle D.A, d \sqcup \exists_x c\rangle \longrightarrow \langle D.B, d'\rangle}{\langle D.\exists^d x A, c\rangle \longrightarrow \langle D.\exists^{d'} x B, c \sqcup \exists_x d'\rangle}$
R5	$\langle D.p(x), c\rangle \longrightarrow \langle D.A, c\rangle \qquad p(x) : -A \in D$

Table 1. The transition system of ccp.

3 Semantic aspects of ccp

In the first few years after its design, ccp had been understood just as a particular case of process algebra. Therefore, the definition of its compositional semantics had been approached by the standard methods, like failure sets and bisimulation. For instance, De Boer et al. [15, 16] used tree-like structures labeled with functions on substitutions. More simple tree-like structures, labeled by constraints, were used by Gabbrielli and Levi [39]. Saraswat and Rinard [56] used similar structures modulo equivalence relations based on bisimulation.

De Boer and Palamidessi [18] realized that, due to the fact that the communication mechanism of ccp is asynchronous, the branching structures used for process algebra are not needed. In fact, which actions are enabled does not depend upon the current state of the environment, but only upon the store. In a transition system this can be made explicit by adding a passive rule that does not exist in the classical concurrent paradigms: an arbitrary assumption of a step made by the environment. This amounts to considering all the possible interactions between the given process and arbitrary environments, and it leads to a simple compositional semantics, consisting of sequences of constraints labeled by assume/tell modes. In this framework the parallel composition corresponds to zip sequences, so that the assumptions of a process match with the actions of the other, and vice-versa.

Independently, a different approach was developed in [57]. The basic idea consists in denoting processes as Scott's closure operators, which have the nice property of being representable by the set of their fixpoints. The operators of the language can then be described as operations on those sets. In particular, parallelism can be modeled simply by intersection.

The semantics developed in [18] and in [57] are based on very different points of view. The one in [18] is more general, in the sense that it applies, without

essential modifications, to many variants of ccp, including the atomic and nondeterministic versions. The one in [57] is very ingenious and elegant, and can be considered one of the principal reasons of the success of ccp. However, it works well only in the basic fragment, the *deterministic eventual tell ccp*, which is obtained from Definition 6 by imposing $n = 1$ in the summation. Both semantics are fully abstract, and therefore in the basic fragment they are equivalent. The precise correspondence was delineated in [19].

One question that had remained open in [18] was how to model infinite computations in an abstract way, i.e. by considering only the limit of the answer substitution. When nondeterminism is present, the denotational characterization of infinite computation is actually a non trivial problem: The semantics based on Smith, Hoare and Plotkin's powerdomains constitute only a partial solution to this problem (in the sense that they identify too much), and the semantics based on metric domains are far from being abstract. This problem was solved in [25] by considering a categorical construction called *Lehmann's powerdomain*, which can be regarded as an extension of Smith's powerdomain. This structure contains more information than the powerdomains, enough to achieve compositionality.

3.1 Analysis and verification

De Boer et al. developed in [20] a system based on the closure operators semantics to prove correctness assertions about concurrent constraint programs. Thanks to the strong properties of ccp, this system is much simpler than the ones developed for other parallel languages. In particular, only the strongest post-condition w.r.t. *True* needs to be considered, and parallel composition is modeled simply by logical conjunction.

Falaschi et al. investigated in [33] various fragments of ccp. Some of them have a very simple semantics based on closure operators. Such semantics can be considered as approximated semantics of ccp, and they were used as a basis for static analysis [32, 34], by means of *abstract interpretation* techniques. These techniques allow to statically optimize programs and to approximate several important semantic properties, such as deadlock detection, groundness propagation etc.

One interesting fragment is *ccp with local choice*: This corresponds in fact to *CLP with delay*, an extension of Constraint Logic Programming which allows efficient implementations. Falaschi et al. [35] and De Boer et al. [23] used this observation for developing the semantics foundations and a verification system of CLP with delay, by means of techniques based on closure operators.

Another approach to the analysis of *ccp* was pursed in [65, 66] where it was extended to *ccp* languages the *generalized semantics* approach to static analysis, initially proposed in [41] for sequential CLP languages. [65] shows that such an extension can be easily achieved for approximations that are closed under anti-entailment: applications include analyses that can identify definite suspensions, e.g., to compute upper bounds to the degree of concurrency in a *ccp* program. For the more common case of entailment closed properties (that are of interest for, e.g., proving suspension freeness), it is shown in [66] that correctness can only be

achieved by modifying the generalized semantics approach so as to introduce a domain-dependent approximation of the synchronization primitive, which cannot be modeled as an entailment test on the abstract domain.

3.2 Fold/unfold transformations of *ccp*

Unfold/fold are source-to source transformation techniques which were first introduced in functional programming by Burstall and Darlington [10], and then adapted to logic programming both for program synthesis and for program specialization and optimization. As shown by a number of applications, these techniques provide a powerful methodology for the development and optimization of large programs, and can be regarded as the basis to be used for partial evaluation.

Despite a large amount of literature in the field of declarative sequential languages, the applications of unfold/fold transformations to concurrent languages are relatively rare. This is partially due to the fact that the nondeterminism and the synchronization mechanisms present in concurrent languages substantially complicate their semantics, thus complicating also the definition of *correct* transformation systems. Nevertheless, these transformation techniques can be very useful also for concurrent languages, since they allow further optimizations related to the simplification of synchronization and communication mechanisms.

One of the few papers addressing this issue is [31], where a transformation system for concurrent constraint programming (*ccp*) was introduced. This systems was inspired by that one of Tamaki and Sato [60], a general framework for the unfold/fold transformation of logic programs, which has remained over the years the main historical reference of the field.

Compared to its predecessors, the system in [31] improves by eliminating the limitation that in a folding operation the *folding rule* has to be non-recursive. Moreover, following de Francesco and Santone [38], the applicability conditions for this operation are based on the notion of "guardedness" and can be checked locally on the program to be folded (rather than on the transformation history). This makes the operation much easier to understand and to implement. Besides folding and unfolding, the transformation system for *ccp* includes several other new operations, namely backward instantiation, ask and tell simplification, branch elimination, conservative ask elimination and distribution. The declarative nature of *ccp* allows one to define reasonably simple applicability conditions for these operations which ensure the total correctness of the system: the original and the transformed program have the same semantics when considering both input/output pairs and (under different applicability conditions) traces, and distinguishing successful, deadlocked, and failed derivations.

From the correctness result follows that the original program is deadlock-free iff the transformed one is, and this allows us to employ the transformation system as an effective tool for proving deadlock-freeness of *ccp* programs. Moreover, the systems allows to optimize programs by eliminating communication channels and synchronization points, by transforming nondeterministic computations into deterministic ones, and by saving of computational *space*. Some of

these improvements were possible already in the context of GHC programs by using the system defined in Ueda and Furukawa [63].

Following the above line of research, [3] investigated transformation techniques based on the *replacement*. This is a powerful operation which can mimic the most common transformation operations such as unfold, fold, switching, distribution. Because of this flexibility, it can be incorrect if used without specific applicability conditions. The above paper presented applicability conditions for ccp and it showed that, under these conditions, the replacement generalizes both the unfolding operation as well as a restricted form of folding operation.

4 Timed Reactive CCP

The *tcc* model is a timed reactive ccp framework introduced by Saraswat et al [53] as an extension of *deterministic* ccp. This model is aimed at programming and modeling timed reactive systems and it elegantly combines deterministic ccp with ideas from the paradigms of Synchronous Languages [2].

In order to increase the specification expressiveness of *tcc*, Nielsen et al [49] introduced a non-deterministic extension of *tcc*, called the *ntcc* calculus. As its predecessor, the *ntcc* calculus takes the view of reactive computation as proceeding in discrete time units (or *time intervals*). Time is conceptually divided into discrete intervals. In each time interval a ccp process receives a *stimulus*, represented as a constraint, from the environment, it executes with this stimulus as the initial store, and when it reaches its resting point, it *responds* to the environment with the final store. Furthermore, the resting point determines a residual process, which is then executed in the next time interval.

As illustrated in [49], this view of reactive computation is particularly appropriate for modeling and programming reactive systems such as robotic devices and micro-controllers. These systems typically operate in a cyclic fashion; in each cycle they receive and input from the environment, compute on this input, and then return the corresponding output to the environment.

4.1 Syntax and Operational Semantics of *ntcc*

The *ntcc* calculus introduces operators to specify temporal executions. The *unit-delay* operation $next\, A$, also present in *tcc*, specifies that A should be executed in the next time interval, and the *unbounded* delay operation $\star A$ specifies that A will be *eventually* executed. The *time-out* operation *unless* c *next* A, also present in *tcc*, specifies that unless c can be inferred from the final store in the current time unit, A should be executed in the next time unit.

Furthermore, to ensure that only terminating processes can be executed within time intervals, procedures are replaced with the simpler replicated form $!A$. The replication operation $!A$ specifies that A will be executed now and in each future time interval. Thus, $!A$ can be viewed as $A \parallel next\, A \parallel next\, (next\, A) \parallel \ldots$

All in all, the agents of *ntcc* include those of ccp in Definition 2 except for procedures, plus the above-mentioned temporal operators. More precisely,

Definition 3. *Assuming a given cylindric constraint system* **C** *the syntax of ntcc* agents *is given by the following grammar:*

$$A ::= stop \mid tell(c) \mid \sum_{i=1}^{n} ask(c_i) \rightarrow A_i \mid A \parallel A \mid \exists x A \\ \mid next\, A \mid \star A \mid unless\ c\ next\, A \mid !A$$

where the c, c_i *are supposed to be* finite constraints *(i.e. algebraic elements) in* $\mathcal{C}$*. For the sake of consistency with Definition 2, an ntcc* process P *can be interpreted as an object of the form* $D.A$ *by decreeing that* $D = \emptyset$*; i.e., the empty set of procedure declarations.*

4.2 Reduction Relations

The operational semantics of *ntcc* is given in terms of an internal reduction relation $\longrightarrow$ given by the rules in Table 1 plus the rules in Table 2 and the observable reduction relation $\Longrightarrow$ given in Table 2.

The *internal* transition $\gamma \longrightarrow \gamma'$ specifies the internal steps much like the ccp transitions $\longrightarrow$ in the previous section. The additional rules **R6-R8** in Table 2 realize the above intuitions about the temporal operators.

The *observable transition* $P \stackrel{(c,d)}{\Longrightarrow} R$ should be read as "P on input c from the environment, reduces in one *time unit* to R and outputs d to the environment". The rule **ROBS** realizes the above intuition by stating that an observable transition from $P = D.A$ labeled by (c, d) is obtained by performing a sequence of internal transitions from the initial configuration $\langle P, c\rangle$ to a final configuration $\langle Q, d\rangle$ with $Q = D.A'$ in which no further internal evolution is possible. The residual process R to be executed in the next time interval is equivalent to $D.F(A')$, where $F(A')$ represents the "future" of A'. The process $F(A')$, given in Definition 4, is obtained by removing from A' summations that did not trigger activity within the current time interval and any local information which has been stored in A', and by "unfolding" the sub-terms within "next" and "unless" expressions. This "unfolding" specifies the evolution across time intervals of processes of the form $next\, B$ and $unless\ c\ next\, B$.

Definition 4 (Future Function). *Let* F *be the partial function defined by*

$$F(A) = \begin{cases} stop & \text{if } A = \sum_{i \in I} c_i \rightarrow A_i \\ F(A_1) \parallel F(A_2) & \text{if } A = A_1 \parallel A_2 \\ \exists x F(B) & \text{if } A = \exists^d x B \\ B & \text{if } A = next\, B \text{ or } A = unless\ c\ next\, B \end{cases}$$

4.3 A simple example of weak pre-emption

In spite of its simplicity, the *tcc* and *ntcc* extensions to ccp are far-reaching. Many interesting temporal constructs can be expressed (see e.g. [53]). For example, *tcc* allows processes to be "clocked" by other processes. This provides meaningful

R6	$\langle D. \star A, d\rangle \longrightarrow \langle D.A^n, d\rangle \qquad n \geq 0$
R7	$\langle D.unless\ c\ next\ A, d\rangle \longrightarrow \langle D.stop, d\rangle,\ d \vdash c$
R8	$\langle D.!A, d\rangle \longrightarrow \langle D.A \parallel next\,!A, c \sqcup d\rangle$
ROBS	$\dfrac{\langle D.A, c\rangle \longrightarrow^* \langle D.A', c'\rangle \not\longrightarrow}{D.A, c \overset{(c,c')}{\Longrightarrow} D.F(A')}$

Table 2. Additional rules for the transitions of *ntcc* processes. The internal reduction $\longrightarrow$ is given by the rules in Table 1 and Rules **R6-R8**. The observable reduction $\Longrightarrow$ is given by Rule **ROBS**. The relation $\longrightarrow^*$ denotes transitive and reflexive closure of $\longrightarrow$. $\gamma \not\longrightarrow$ holds iff that is no γ' such that $\gamma \longrightarrow \gamma'$. The function F is given in Definitions 4

pre-emption constructs and the ability to define *multiple forms of time* instead of only having a unique global clock.

A rather simple example is the specification of a power-saver:

$$A = !\ unless\ (LightsOff)\ next\ \star\ tell(LightsOff)$$

The power-saver agent A runs forever, hence it is replicated. Furthermore, unless A can infer that the lights are already off in the current time interval, A should turn them off either in the next time unit or sometime later.

Notice that because of the weak pre-emption nature of the time-out operation in *ntcc*, it is not possible to specify that the lights should be turned off within the current time interval unless they are already off.

The work in [54] introduces *Default tcc* as an extension of *tcc* with the ability to define strong pre-emption. In this model, the time-out operation can trigger activity in the current time interval. Strong pre-emption is useful when an action must be triggered immediately on the absence of a constraint c rather than delayed to the next interaction.

4.4 Observables and their Characterizations

Let us consider an infinite sequence of observable transitions:

$$P \ = \ P_1 \overset{(c_1,c_1')}{\Longrightarrow} P_2 \overset{(c_2,c_2')}{\Longrightarrow} P_3 \overset{(c_3,c_3')}{\Longrightarrow} \ldots$$

Intuitively, at time interval i, with $i \geq 0$, the process P_i gets a stimulus c_i and then it provides a response c_i' and evolves into P_{i+1}. We shall also represent this run as $P \overset{(\alpha,\alpha')}{\Longrightarrow}$ where $\alpha = c_1.c_2.c_3.\ldots$ and $\alpha' = c_1'.c_2'.c_3'\ldots$.

The observable *input-output* behaviour of an *ntcc* process is its set of stimulus-response sequences. The *strongest-postcondition*, or *quiescent behaviour*, of a process P is the set of sequences on input of which P can run without adding any information whatsoever. More precisely,

Definition 5 (Observables of *ntcc*). *Let P be a process. The input-output behaviour of P is given by $\mathcal{O}_{io}(P) = \{(\alpha, \alpha') \mid P \xrightarrow{(\alpha,\alpha')}\}$. The strongest post-condition of P is given by $\mathcal{O}_{sp}(P) = \{\alpha \mid P \xrightarrow{(\alpha,\alpha)}\}$.*

As shown in [49] the observable input-output behaviour of deterministic *ntcc* processes (i.e., *tcc* processes) can be compositionally specified as closure operators over sequences of constraints much like for the deterministic ccp case. Also, by building on the strongest-postcondition semantics for ccp in [20], the work in [49] includes a compositional characterization of the quiescent behaviour of *ntcc* processes as well as a proof system for their temporal properties. The *ntcc* proof system is similar to Dijkstra's proof system for the strongest postcondition of imperative programs.

In [48] the authors provided a hierarchy of *ntcc* variants based on the input-output behaviour. A variant C is said to be *as expressive as* a variant C' if for every process P in C', one can compute a process $E(P)$ in C such that $\mathcal{O}_{io}(P) = \mathcal{O}_{io}(E(P))$. The variants were obtained by replacing replication with alternative mechanisms to specify infinite behaviour: Namely, procedure definitions, static-scoping parameterless recursion, and dynamic-scoping parameterless recursion. It was shown that *ntcc* is equally expressive to the variant with static-scoping parameterless recursion. These variants were also shown to be strictly less expressive than the variant with parametric procedures which in turn was shown to be equally expressive to the variant with dynamic-scoping parameterless recursion. The authors also showed that the input-output behavior of every *ntcc* processes is *omega-regular*; i.e. it can be specified by a finite-state Büchi automaton [9].

In [36] it is defined a framework for the declarative debugging of *ntcc* programs, which is based on a fixpoint semantics for this language. A general framework, parametric w.r.t an abstract domain, for the static analysis of tcc programs is provided in [37].

5 Another timed *ccp* language

A different timed extension of *ccp*, called *tccp*, was proposed in [21]. Similarly to the previously mentioned timed languages (*tcc*) [53] and *default tcc* [54], *tccp* is a language for reactive programming where computation takes a bounded period of time rather than being instantaneous (as it is in ESTEREL [2]). However, differently from *tcc* and *default tcc*, which are inspired by the deterministic synchronous languages, *tccp* follows the guidelines of the timed process algebras approach and allows for non-determinism. This corresponds to a different view and use of a timed language: deterministic languages can be used for programming "kernels" of real-time systems, since deterministic systems are simpler to specify, debug and analyze. However, non-determinism arises when considering larger reactive systems involving several processes running on different processors and communicating via asynchronous links. These (timed) systems can be naturally specified and programmed by using a non-deterministic language.

Indeed all the existing timed process algebras and almost all the variants of Statecharts admit non-determinism.

Notice that the *ntcc* calculus discussed in the previous section, is also a non-deterministic timed ccp language. However, *ntcc* is an orthogonal non-deterministic extension of *tcc*, while *tccp* is an orthogonal timed nondeterministic extension of *ccp*. That means that, unlike in *tccp*, in *ntcc* computation proceeds as in the synchronous languages.

Below we first describe the *tccp* language and its operational semantics. Then we define a fix-point semantics for it which is based on reactive sequences and which is fully abstract w.r.t. the input/output notion of observables. All the technical definitions and results in this section are from [21].

5.1 Syntax and operational semantics of *tccp*

When querying the store for some information which is not present (yet) a *ccp* agent will simply suspend until the required information has arrived. In many applications involving time, however, often one cannot wait indefinitely for an event. Consider for example the case of a bank teller machine: if there is a problem with the authorization of the bank, after a reasonable amount of time the card should be given back to the customer. In order to model such a situation then the language should allow us to specify that, in case a given time bound is exceeded (i.e. a time-out occurs), the wait is interrupted and an alternative action is taken. Moreover, in some cases it is also necessary to abort an active process A and to start a process B when a specific event occurs (this is usually called *preemption* of A). For example, according to a typical pattern, A is the process controlling the normal activity of some physical device, the event indicates some abnormal situation and B is the exception handler.

In order to enrich *ccp* agents with such timing mechanisms, we introduce a *discrete global clock* and assume that *ask and tell actions take one time-unit*. Computation evolves in steps of one time-unit, so called clock-cycles, and action prefixing is the syntactic marker which distinguishes a time instant from the next one.

Furthermore, we make the assumption that parallel processes are executed on different processors, which implies that at each moment every enabled agent of the system is activated. This assumption, which is common to many timed process algebras, gives rise to what is called *maximal parallelism*.

Since the store is monotonically increasing and one can have dynamic process creation, clearly the previous assumptions in principle imply that the constraint solver takes a constant time (no matter how big the store is) and that there is an unbound number of processors. In practice, however, one can impose suitable restrictions on programs, thus ensuring that the (significant part of the) store and the number of processes do not exceed a fixed bound.

In order to express *time-out* and *preemption* which, as previously mentioned, are essential to many applications, the language is enriched by introducing a more basic timing construct of the form

$$now\ c\ then\ A\ else\ B.$$

This construct is similar to the analogous one used in [53], even though here it has a different interpretation: If c is entailed by the store then the above agent behaves as A at the *current* time instant, otherwise it behaves as B (at the current time instant). Note that the ability to detect the *absence* of an event is essential here.

Thus, we end up with the following syntax.

Definition 6 (*tccp* Language). *Assuming a given cylindric constraint system* **C** *the syntax of agents is given by the following grammar:*

$$A ::= stop \mid tell(c) \mid \sum_{i=1}^{n} ask(c_i) \rightarrow A_i \mid now\ c\ then\ A\ else\ B \mid A \parallel B \mid \exists x\ A \mid p(x)$$

where the c, c_i are supposed to be finite constraints (i.e. algebraic elements) in $\mathcal{C}$. A tccp process P is then an object of the form $D.A$, where D is a set of procedure declarations of the form $p(x) :- A$ and A is an agent.

In order to simplify the notation, in the following we will omit the $\sum_{i=1}^{n}$ whenever $n = 1$ and we will use $tell(c) \rightarrow A$ as a shorthand for $tell(c) \parallel (ask(true) \rightarrow A)$.

The operational model of *tccp* can be formally described by a transition system $T = (Conf, \longrightarrow)$ where we assume that each transition step takes exactly one time-unit. Configurations (in) *Conf* are pairs consisting of an agent and a constraint in $\mathcal{C}$ representing the common *store*. The transition relation $\longrightarrow \subseteq Conf \times Conf$ is the least relation satisfying the rules **R1, R2, R4 and R5** in Table 1 plus the rules in Table 3.

Notice that the rules now characterizes also the temporal evolution of the system, so $\langle A, c\rangle \longrightarrow \langle B, d\rangle$ means that if at time t we have the agent A and the store c then at time $t + 1$ we have the agent B and the store d.

In particular, Rule **R1** (in Table 1) shows that the evaluation of a tell action takes one time-unit, thus the updated store $c \sqcup d$ will be visible only starting from the next time instant. Analogously, also the evaluation of an ask action takes one time-unit (rule **R2**).

Let us now briefly discuss the new rules in Table 3.

Rules **R3bis** and **R3ter**, which replace rule **R3** of Table 1, model the parallel composition operator in terms of *maximal parallelism*: The agent $A \parallel B$ executes in one time-unit all the initial enabled actions of A and B.

The rules **R9**-**R12** show that the agent *now c then A else B* behaves as A or B depending on the fact that c is or is not entailed by the store. Note that here, differently from the case of the ask, the evaluation of the guard is instantaneous. Since A and B could contain nested *now then else* agents, a limit for the number of these nested agents should be fixed. However, for recursive programs such a limit is ensured by the presence of the procedure call, since we assume that the evaluation of such a call takes one time unit.

Using the transition system described by (the rules in) Table 1 we can define the following notion of observables which considers the input/output of terminating computations, including the deadlocked ones. Here and in the sequel $\longrightarrow^*$ denotes the reflexive and transitive closure of the relation $\longrightarrow$.

R3bis	$\dfrac{\langle A, c\rangle \longrightarrow \langle A', c'\rangle \quad \langle B, c\rangle \longrightarrow \langle B', d'\rangle}{\langle A \parallel B, c\rangle \longrightarrow \langle A' \parallel B', c' \sqcup d'\rangle}$	
R3ter	$\dfrac{\langle A, c\rangle \longrightarrow \langle A', c'\rangle \quad \langle B, c\rangle \not\longrightarrow}{\begin{array}{c}\langle A \parallel B, c\rangle \longrightarrow \langle A' \parallel B, c'\rangle \\ \langle B \parallel A, c\rangle \longrightarrow \langle B \parallel A', c'\rangle\end{array}}$	
R9	$\dfrac{\langle A, d\rangle \longrightarrow \langle A', d'\rangle}{\langle \mathit{now}\ c\ \mathit{then}\ A\ \mathit{else}\ B, d\rangle \longrightarrow \langle A', d'\rangle}$	$d \vdash c$
R10	$\dfrac{\langle A, d\rangle \not\longrightarrow}{\langle \mathit{now}\ c\ \mathit{then}\ A\ \mathit{else}\ B, d\rangle \longrightarrow \langle A, d\rangle}$	$d \vdash c$
R11	$\dfrac{\langle B, d\rangle \longrightarrow \langle B', d'\rangle}{\langle \mathit{now}\ c\ \mathit{then}\ A\ \mathit{else}\ B, d\rangle \longrightarrow \langle B', d'\rangle}$	$d \nvdash c$
R12	$\dfrac{\langle B, d\rangle \not\longrightarrow}{\langle \mathit{now}\ c\ \mathit{then}\ A\ \mathit{else}\ B, d\rangle \longrightarrow \langle B, d\rangle}$	$d \nvdash c$

Table 3. The additional rules for *tccp*.

Definition 7 (Observables). *Let A be an agent. We define $\mathcal{O}_{io}(A) = \{\langle c, d\rangle \mid \langle A, c\rangle \longrightarrow^* \langle B, d\rangle \not\rightarrow\}$.*

5.2 Programming example

We show now how some typical reactive programming idioms can be derived from the basic combinators of *tccp*. Then we use these in a programming example.

Time-out The timed guarded choice agent

$$\sum_{i=1}^{n} \mathit{ask}(c_i) \rightarrow A_i \ \mathit{time\text{-}out(m)}\ B$$

waits at most m time-units ($m \geq 0$) for the satisfaction of one of the guards. Before this time-out the process behaves just like the guarded choice: As soon as there exist enabled guards, one of them and the corresponding branch is nondeterministically selected. After waiting for m time-units, if no guard is enabled, the timed choice agent behaves as B. This agent can be defined inductively as follows. Let us denote by A the agent $\sum_{i=1}^{n} \mathit{ask}(c_i) \rightarrow A_i$. In the base case, $m = 0$, we define $\sum_{i=1}^{n} \mathit{ask}(c_i) \rightarrow A_i$ time-out(0) B as the agent

$$\begin{array}{l} \mathit{now}\ c_1\ \ \mathit{then}\ A\ \ \mathit{else} \\ \quad (\ \mathit{now}\ c_2\ \ \mathit{then}\ A\ \ \mathit{else} \\ \qquad \vdots \\ \qquad\quad (\ \mathit{now}\ c_n\ \mathit{then}\ A\ \ \mathit{else}\ \mathit{ask}(\mathit{true}) \rightarrow B) \ldots) \end{array}$$

For the inductive step we define $\sum_{i=1}^{n} ask(c_i) \rightarrow A_i \;\; time\text{-}out(m)\; B$ as

$$\sum_{i=1}^{n} ask(c_i) \rightarrow A_i \;\; time\text{-}out(0) \left(\sum_{i=1}^{n} ask(c_i) \rightarrow A_i \;\; time\text{-}out(m\text{-}1) \;\; B \right).$$

Watchdogs These are typical preemption primitives of such languages as ESTEREL and are used to interrupt the activity of a process on signal from a specific event. Since events are expressed by constraints, a watchdog can be defined as the process

$$do\ A\ watching\ c else B$$

which behaves as A, as long as c is not entailed by the store; when c is entailed, the process A is immediately aborted and process B is started. We have here a form of weak preemption in which the abortion of A is performed in the next time interval. In fact, even though A is aborted at the *same* time instant of the detection of the entailment of c, if c is detected at time t then c has to be produced at time t' with $t' < t$.

Previous watchdog agent can be defined (by induction on the structure of process A) in terms of the other constructs of the language (see [21]). For example in case of the *tell* process one has the following translation

$$do\ tell(d)\ watching\ c\ else\ B \;\Rightarrow now\ c\ then\ B\ else\ tell(d),$$

As a simple example of a *tccp* program let us now consider a system *s(Ex)* consisting of two processes *p1* and *p2* which perform some time critical activities, reacting to external inputs transmitted on the channel *Ex*. The system is continuously checked by a controller which receives a stream of *ok* messages by each process *pi*. Each *ok* message is sent at unpredictable time instants, however it is assumed that each *pi* is working correctly iff it sends the next ok within n time-units from the previous one. When this limit is exceeded by a process *pi* the controller aborts the whole system, starts a recovery routine *rr* for the activity of *pi* and then restart the system. The system *s(Ex)* is implemented by the following program where the specific tasks of the *pi*'s and of the recovery routines are not specificed:

s(Ex):- $\exists$ *Alarm,O1,R1,O2,R2*
((do p1(Ex,O1,R1) $\parallel$ *p2(Ex,O2,R2) watching Alarm = on)*
$\parallel$ *controller(O1,O2,R1,R2))*

controller(O1,O2,R1,R2):- $\exists$ *A1,A2*
(do c(O1,A1) $\parallel$ *c(O2,A2) watching Alarm = on else*
(now (A1 = on $\sqcup$ *A2 = on) then rr(R1)* $\parallel$ *rr(R2) else*
now A1 = on then rr(R1) else
now A2 = on then rr(R2))
$\parallel$ *restart(Ex))*

c(O,A):- *ask (*$\exists$ *Y.O=[ok|Y])* $\rightarrow$ *(*$\exists$ *Y tell(O=[ok|Y])* $\rightarrow$*c(Y,A))*
timeout(n) tell(Alarm = on $\sqcup$ *A = on)*

5.3 The denotational model

It is easy to see that the operational semantics which associates to an agent A its observables $\mathcal{O}_{io}(A)$ is not compositional. A compositional characterization of the operational semantics can be obtained by using sequences of pairs of finite constraints, so called *timed reactive sequences*, analogous to those that we have seen in the semantics of *ccp*.

However, a reactive sequence is now provided with a different interpretation which accounts for the timing aspects. In fact such a sequence has the form

$$\langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle \langle d, d\rangle$$

and each pair of constraints $\langle c_i, d_i\rangle$ now represents a computation step performed by the agent A which, at time i, assuming c_i as input constraint produces the constraint d_i. The last pair denotes a "stuttering step" in which no further information can be produced by the agent, thus indicating that a "resting point" has been reached.

Since in *tccp* computations the store evolves monotonically and the constraints arising from computation steps are finite, it is natural to assume that reactive sequences are monotonically increasing and contains only finite constraints. The set of all reactive sequences is denoted by $\mathcal{S}$ and its typical elements by $s, s_1 \ldots$, while sets of reactive sequences are denoted by $S, S_1 \ldots$ and ε indicates the empty reactive sequence. The semantics R which associates to an agent the reactive sequences that it generates can be defined by a fixpoint construction as follows.

Definition 8. *The semantics $R \in Agent \to \mathcal{P}(\mathcal{S})$ is defined as the least fixed-point of the operator $\Phi \in (Agent \to \mathcal{P}(\mathcal{S})) \to Agent \to \mathcal{P}(\mathcal{S})$ defined by*

$$\begin{aligned}\Phi(I)(A) = &\{\langle c, d\rangle \cdot w \in \mathcal{S} \mid c \in \mathcal{C}, \langle A, c\rangle \to \langle B, d\rangle \text{ and } w \in I(B)\}\\ &\cup\\ &\{\langle c, c\rangle \cdot w \in \mathcal{S} \mid \langle A, c\rangle \not\to \text{ and } w \in I(A) \cup \{\varepsilon\}\}.\end{aligned}$$

The ordering on $Agent \to \mathcal{P}(\mathcal{S})$ is that of (point-wise extended) set-inclusion and it is straightforward to check that Φ is continuous, so standard results allows us to construct the least fixpoint in ω steps.

It is possible to show that the above semantics is correct (w.r.t. the input/ouput observables) and compositional, however is not fully abstract, since it distinguishes *tccp* agents whose observables are the same under any possible context. In order to obtain a fully abstract model one needs to introduce a suitable abstraction on traces, however, due to the presence of the *now then else* construct and of maximal parallelism, one cannot use here the abstraction which has been used in [24] for *ccp* since this would be incorrect (it would identify agents which can be distinguished by a context). This semantic difference has also an expresiveness counterpart, indeed one can show [21] that *tccp* is strictly more expressive than. *ccp*.

So, the full abstraction problem for *tccp* cannot be reduced to that one for *ccp*. Indeed, differently from the case of ccp, the definition of a fully abstract

semantics for *tccp* requires the ability to specify the "difference" $c_i \setminus d_{i-1}$ between an assumption c_i (at time i) and the previous contribution d_{i-1} (at time $i-1$). Such a difference is formalized by using the algebraic notion of weak relative pseudo-complement [42, 4]. Using this difference the abstraction α on set of sequences can be defined as follows.

Definition 9 (Abstraction). *Let s, s' be reactive sequences. Then the $\preceq$ relation is defined as follows:*

- *$s \preceq s'$ iff for some sequences s_1 and s_2 one has that $s = s_1 \cdot \langle a, b\rangle\langle c, d\rangle \cdot s_2$, $s' = s_1 \cdot \langle a, b'\rangle\langle c, d\rangle \cdot s_2$ and $(c \setminus b') \leq (c \setminus b)$.*

Moreover the (equivalence) relation $\simeq$ is defined as follows

- *$s \simeq s'$ iff the sequences s and s' differ only in the number of repetitions of the last element.*

Given a set of reactive sequences S, $\alpha(S)$ denotes the least set S' such that the following holds:

(i) *$S \subseteq S'$,*
(ii) *if $s' \in S'$ and either $s \preceq s'$ or $s \simeq s'$, then $s \in S'$.*

The fully abstract semantics R_α is obtained by simply applying the function α to $R(A)$. One can show that the semantics obtained in this way is compositional (w.r.t. all the operators of the language) and correct (since it allows to reconstruct the observables $\mathrm{O}_{io}(A)$). Moreover it is also fully abstract, as shown by the following theorem.

Theorem 1 (Full abstraction). *Assume that the constraint system is weakly relative pseudo-complemented. Then, for any pair of tccp agents A and B, $\alpha(R(A)) = \alpha(R(B))$ iff $\mathcal{O}_{io}(C[A]) = \mathcal{O}_{io}(C[B])$ for each context $C[\cdot]$.*

FInally it is worth noting that a temporal logic for reasoning on *tccp* programs, inpired by this semantics, has been defined in [22].

6 Other extensions of *ccp*

In this section we survey some more recent extensions of *ccp* which mainly deal with probabilistic and uncertainty aspects.

6.1 Probabilistic *ccp*

In [27] the concurrent constraint programming paradigm is extended with a probabilistic choice construct which replaces the nondeterministic choice of the original paradigm; this allows a program to make stochastic moves during its execution, so that it may be seen as a stochastic process. This embedding of randomness within the semantics of a well structured programming paradigm,

like *ccp*, also aims at providing a sound framework for formalising and reasoning about randomised algorithms and programs. For the resulting language called probabilistic *ccp*, a fixpoint semantics is given in [26, 28], which is based on vector spaces and the Brouwer's fixpoint theorem. The addition of probabilities allows for a natural formulation of the average behaviour of a program, whose specification and analysis is particularly important in the study of system performance and reliability. It also allows for an average-case analysis of programs as opposite to the worst case analysis common to the classical static analysis approaches [30].

Concurrent Constraint Programming has been used as a reference programming paradigm for the introduction of a general theory of probabilistic abstract interpretation, which re-formulates the classical theory of abstract interpretation in a setting suitable for a quantitative reasoning about programs. In this setting, linear spaces replace the classical order-theoretic domains, and the notion of the so-called *Moore-Penrose pseudo-inverse* of a linear operator replaces the classical notion of a Galois connection. The resulting abstractions turn out to be *close* approximations of the concrete semantics, so that closeness becomes a quantitative replacement for classical safety [29].

6.2 *ccp* for Service Level Agreement

Service Oriented Computing is an emerging paradigm that builds upon the notion of services as interoperable elements that can be described, published, searched and composed. Services may expose both functional properties (i.e. what they do) and non-functional properties (i.e. the way they are supplied). A Service Level Agreement (SLA) is a contract between two parties, usually a service provider and a customer, that records non-functional properties about a service like performance, availability, and cost.

Recently several extensions of the pure *ccp* language have been proposed for dealing with Service Level Agreement aspects. Here we briefly describe the main proposals in this area.

The concurrent constraint pi-calculus (cc-pi calculus) [12] is a model of Service Level Agreement negotiations that is inspired by both *ccp* and name-passing calculi. Specifically, the cc-pi calculus combines basic operations of concurrent constraint programming, such as *ask* and *atomic tell*, with a symmetric, synchronous mechanism of interaction between senders and receivers, where the sent name is 'fused' (i.e. identified) to the received name and such an *explicit fusion* enables using interchangeably the two names. The cc-pi calculus is parametric with respect to the choice of an underlying constraint system that is defined using a suitable semiring structure, equipped with a notion of names. Moreover, cc-pi includes a restriction operation that allows for local stores of constraints. Synchronisations of interacting processes may have the effect of combining local into global stores.

Some semantic aspects of the cc-pi calculus are studied in [13], where its is defined a notion of open bisimilarity *à la* pi-calculus for cc-pi. Essentially, two processes are open bisimilar if they have the same stores of constraints - which

can be statically checked - and if their moves can be mutually simulated. In [13] it is also shown that the polyadic Explicit Fusion calculus introduced by Gardner and Wischik can be translated into monadic cc-pi and such a transition preserves open bisimilarity.

In [11] a further extension of the cc-pi calculus is defined by including primitives for distributed nested commits, inspired by the cjoin calculus (introduced by Bruni, Melgratti, and Montanari). The two key operations of cjoin are: the 'abort with compensation', to stop a negotiation and activate a compensating process, and the 'commit', to store a partial agreement among the parties before moving to the next negotiation phase. This extended cc-pi calculus comes equipped with both a small- and a big-step operational semantics which are proved to coincide.

A different line of research is focused on the use of, so called soft constraint, to model qualitative aspects of Service Level Agreement in the context of the *ccp* paradigm. As described in more detail in another chapter of this book [40], soft constraints extend classical constraints to represent multiple consistency levels, and thus provide a way to express preferences, fuzziness, and uncertainty. An extension of the *ccp* framework which allows soft constraints in the calculus has been proposed in [6]. In this extension it is permitted to add (tell) or check (ask) for soft constraints and the language is enriched with tell/ask thresholds which can express the level of consistency of the store, thus allowing to prune and direct the search for a solution (when some consistency levels are not satisfied). The resulting language, called soft cc (scc), can be also very useful in many web-related scenarios, since allows web agents to express their interaction and negotiation protocols, and also to post their requests in terms of preferences. Differently from the case of "hard" (or "crisp") constraints, the underlying soft constraint solver here can find an agreement among the agents even if their requests are incompatible.

A timed extension of scc has been proposed in [5] in order to be able to express also Quality of Service aspects which involve time. As in the case of scc, tell and ask agents are equipped with a preference (or consistency) threshold which is used to determine their success or suspension. The time and the semantic model of this extension follows the lines of the tccp language presented in Section 5.

Another extension of scc, which allows the nonmonotonic evolution of the constraint store, is defined in [7]. To accomplish this, some new operations are introduced: the retract(c) reduces the current store by c; the updateX(c) transactionally relaxes all the constraints of the store that deal with the variables in X set, and then adds a constraint c; the nask(c) tests if c is not entailed by the store. This language allows the management of resources that need a given Quality of Service: the requirements of all the parties should converge, through a negotiation process (which involves retract of information), on a formal agreement defined as the Service Level Agreement, which specifies the contract that must be enforced.

7 Some working ccp systems

In this section we shall briefly survey some existing working ccp systems.

The programming language *jcc* [55] was designed as an integration of *default tcc* into Java and is intended for embedded reactive systems. In `jcc` users can define their own constraint system and thus specialize the language to particular domains. The main purpose of *jcc* is to provide a model of loosely-coupled concurrent programming in Java. The language introduces the notion of a *vat*. A *vat* can be thought of as encapsulating a single synchronous, reactive *tcc* computation. A computation consists of a dynamically changing collection of interacting *vats*, communicating with each other through shared, mutable objects called *ports*. Asking and telling objects can read from and write into the port, respectively and the temporal constructs from the underlying *tcc* model allow an object to specify code whose execution should be delayed.

In the *hybrid concurrent constraint programming* language, *hcc* [43], it is possible to express discrete and continuous evolution of time. More precisely, there are points at which discontinuous change may occur (i.e. the execution proceeds as burst of activity) and open intervals in which the state of the system changes continuously (i.e. the system evolves continuously and autonomously). The notion of *continuous constraint system* (a real-time extension of constraint systems) is introduced to describe the continuous evolution. The syntax of *hcc* extends that of *tcc* with the construct *hence* P, asserting that P holds continuously beyond the current instant. An interpreter of *hcc* can be found at http://www-cs-students.stanford.edu/∼vgupta/hcc/hcc.html.

NtccSim is a simulation tool developed in Oz for *ntcc*, one of the temporal models previously described . Constraints over finite domains and real intervals have been used to implement models of biological systems. *NtccSim* can be found at http://cic.javerianacali.edu.co/wiki/doku.php?id=grupos:avispa:ntccsim. An implementation of the other temporal model previously described, *tccp*, can be found at http://users.dsic.upv.es/∼villanue/tccpInterpreter.

The LMNTAL model [64] provides a scalable, uniform view of concurrent programming concepts such as processes, messages, synchronous and asynchronous computation. It inherits ideas from the concurrent constraint language GHC and from Janus. Communication is based on constraints over logical variables. Processes sharing variables are thought of as been connected. Multisets of nested nodes and links are a first-class notion in LMNtal. Transformations are rules, much like in Janus. LMNtal provides both channel mobility and process mobility: it allows dynamic reconfiguration of process structures as well as the migration of nested computations. An implementation of LMNtal can be found at http://www.ueda.info.waseda.ac.jp/lmntal/.

CORDIAL [50] is a visual language intended as a user transparent integration of constraints and objects. The language is based on a ccp calculus extended with the notion of objects and classes. Methods are represented as windows. Objects within methods are represented by closed contours. Object methods launch ccp processes that, in addition to the usual ask and tell operations, can send messages to other objects. Messages are objects connected by links to object

mailboxes. Objects are identified by an associated constraint parametrized on a local variable (so-called *self*). Senders willing to invoke some object method post a constraint involving some variable, say X, and then send the message to X. Any object such that its associated constraint can be entailed by the store conjoined with the constraint $self = X$, is eligible to accept the message. Some eligible object is then non-deterministically chosen to handle the message. This scheme allows very complex patterns of communication and mobility.

References

1. J. Bergstra and J. Klop. Process algebra: specification and verification in bisimulation semantics. In *Mathematics and Computer Science II*, CWI Monographs, pages 61 – 94. North-Holland, 1986.
2. G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.
3. M. Bertolino, S. Etalle, and C. Palamidessi. The replacement operation for CCP programs. In A. Bossi, editor, *Proceedings of LOPSTR*, volume 1817 of *Lecture Notes in Computer Science*, pages 216–233. Springer, 1999.
4. G. Birkhoff. Lattice theory. *AMS Colloquium Publications*, XXV, 1967.
5. S. Bistarelli, M. Gabbrielli, M. C. Meo, and F. Santini. Timed soft concurrent constraint programs. In D. Lea and G. Zavattaro, editors, *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 2008.
6. S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. *ACM Trans. Comput. Log.*, 7(3):563–589, 2006.
7. S. Bistarelli and F. Santini. A nonmonotonic soft concurrent constraint language for sla negotiation. *Electr. Notes Theor. Comput. Sci.*, 236:147–162, 2009.
8. S. Brookes, C. Hoare, and W. Roscoe. A theory of communicating sequential processes. *Journal of ACM*, 31:499–560, 1984.
9. J. R. Buchi. On a decision method in restricted second order arithmetic. In *Proc. Int. Cong. on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.
10. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977.
11. M. Buscemi and H. Melgratti. Transactional service level agreements. In *Post-Proceedings of TGC 2007, 3rd Symposium on Trustworthy Global Computing*, volume 4912 of *Lecture Notes in Computer Science*, pages 124–139. Springer-Verlag, 2008.
12. M. Buscemi and U. Montanari. Cc-pi: A constraint-based language for specifying service level agreements. In *Proceedings of ESOP 2007, 16th European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 18–32. Springer-Verlag, 2007.
13. M. Buscemi and U. Montanari. Open bisimulation for the concurrent constraint pi-calculus. In *Proceedings of ESOP 2008, 17th European Symposium on Programming*, volume 4912 of *Lecture Notes in Computer Science*, pages 254–268. Springer-Verlag, 2008.
14. K. Clark and S. Gregory. PARLOG: parallel programming in logic. *ACM Trans. on Programming Languages and Systems*, 8(1):1–49, 1986.

15. F. de Boer, J. Kok, C. Palamidessi, and J. Rutten. Control flow versus logic: a denotational and a declarative model for Guarded Horn Clauses. In A. Kreczmar and G. Mirkowska, editors, *Proc. of the Symposium on Mathematical Foundations of Computer Science*, volume 379 of *Lecture Notes in Computer Science*, pages 165–176. Springer-Verlag, 1989.
16. F. de Boer, J. Kok, C. Palamidessi, and J. Rutten. Semantic models for a version of PARLOG. In G. Levi and M. Martelli, editors, *Proc. of the Sixth International Conference on Logic Programming*, Series in Logic Programming, pages 621–636, Lisboa, 1989. The MIT Press. Extended version in [17].
17. F. de Boer, J. Kok, C. Palamidessi, and J. Rutten. Semantic models for Concurrent Logic Languages. *Theoretical Computer Science*, 86(1):3 – 33, 1991. A short version appeared on Proceedings of the Seventh International Conference on Logic Programming, Lisboa, 1989.
18. F. de Boer and C. Palamidessi. A Fully Abstract Model for Concurrent Constraint Programming. In S. Abramsky and T. Maibaum, editors, *Proc. of TAPSOFT/CAAP*, volume 493 of *Lecture Notes in Computer Science*, pages 296–319. Springer-Verlag, 1991.
19. F. de Boer and C. Palamidessi. On the semantics of concurrent constraint programming. In K. Broda, editor, *Proc. of ALPUK 92*, Workshops in Computing, pages 145 – 173. Springer-Verlag, 1992.
20. F. S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM Transactions on Programming Languages and Systems*, 19(5):685–725, 1997.
21. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A timed concurrent constraint language. *Inf. Comput.*, 161(1):45–83, 2000.
22. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A temporal logic for reasoning about timed concurrent constraint programs. In *TIME*, pages 227–233, 2001.
23. F. S. de Boer, M. Gabbrielli, and C. Palamidessi. Proving correctness of constraint logic programs with dynamic scheduling. In R. Cousot and D. A. Schmidt, editors, *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings*, volume 1145 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 1996.
24. F. S. de Boer and C. Palamidessi. On the asynchronous nature of communication in concurrent logic languages: A fully abstract model based on sequences. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR*, volume 458 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 1990.
25. F. S. de Boer, A. D. Pierro, and C. Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, 151(1):37–78, 1995.
26. A. Di Pierro and H. Wiklicky. A Banach Space Based Semantics for Probabilistic Concurrent Constraint Programming. In X. Lin, editor, *Proc. 4th Australasian Theory Symposium, CATS'98*, volume 20 – 3 of *Australian Computer Science Communications*, pages 245–259, Singapore, 1998. Springer Verlag.
27. A. Di Pierro and H. Wiklicky. An Operational Semantics for Probabilistic Concurrent Constraint Programming. In Y. C. P. Iyer and D. Schmidt, editors, *Proc. ICCL'98 – International Conference on Computer Languages*, IEEE Computer Society and ACM SIGPLAN, pages 174–183, Chicago, 1998. IEEE Computer Society Press.
28. A. Di Pierro and H. Wiklicky. Probabilistic Concurrent Constraint Programming: Towards a Fully Abstract Model. In G. Brim and Zlatuska, editors, *Proc. of*

the 23rd International Symposium on Mathematical Foundations of Computer Science, MFCS'98, Lecture Notes in Computer Science, Brno, Czech Republic, 1998. Springer Verlag.
29. A. Di Pierro and H. Wiklicky. Concurrent Constraint Programming: Towards Probabilistic Abstract Interpretation. In M. Gabbrielli and F. Pfenning, editors, *Proceedings of PPDP'00 – Priciples and Practice of Declarative Programming*, pages 127–138, Montréal, Canada, September 2000. ACM SIGPLAN, Association of Computing Machinery.
30. A. Di Pierro and H. Wiklicky. Quantitative observables and averages in Probabilistic Concurrent Constraint Programming. In K. Apt, T. Kakas, E. Monfroy, and F. Rossi, editors, *New Trends in Constraints — Selected Papers of the ERCIM/Compulog Workshop on Constraints, October 1999, Paphos, Cyprus*, number 1865 in Lecture Notes in Computer Science, pages 212–236, Berlin — Heidelberg — New York, 2000. Springer Verlag.
31. S. Etalle, M. Gabbrielli, and M. C. Meo. Transformations of ccp programs. *ACM Trans. Program. Lang. Syst.*, 23(3):304–395, 2001.
32. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Compositional analysis for concurrent constraint programming. In *Proc. of the Eight Annual IEEE Symposium on Logic in Computer Science*, pages 210–221, Los Alamitos, California, 1993. IEEE Computer Society Press.
33. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence in concurrent constraint programming. *Theoretical Computer Science*, 183(2):281–315, 1997.
34. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence in concurrent constraint programming. *Theoretical Computer Science*, 183(2):281–315, 1997.
35. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Constraint Logic Programming with Dynamic Scheduling: A Semantics Based on Closure Operators. *Information and Computation*, 137(1):41–67, 1997.
36. M. Falaschi, C. Olarte, C. Palamidessi, and F. Valencia. Declarative diagnosis of temporal concurrent constraint programs. In *Proc. of ICLP'07*. Springer LNCS 4670, 2007.
37. M. Falaschi, C. Olarte, and F. Valencia. A framework for abstract interpretation of timed concurrent constraint programs. In *Proc. of PPDP'09*, pages 107–118. ACM Sigplan, 2009.
38. N. D. Francesco and A. Santone. Unfold/fold transformations of concurrent processes. In H. Kuchen and S. D. Swierstra, editors, *PLILP*, volume 1140 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 1996.
39. M. Gabbrielli and G. Levi. Unfolding and fixpoint semantics for concurrent constraint logic programs. In H. Kirchner and W. Wechler, editors, *Proc. of the Second Int. Conf. on Algebraic and Logic Programming*, Lecture Notes in Computer Science, pages 204–216, Nancy, France, 1990. Springer-Verlag.
40. M. Gavanelli and F. Rossi. *Constraint Logic Programming*, volume 6000 of *Lecture Notes in Computer Science*, chapter 3. Springer-Verlag, 2010.
41. R. Giacobazzi, S. K. Debray, and G. Levi. Generalized semantics and abstract interpretation for constraint logic programs. *Journal of Logic Programming*, 25(3):191–247, 1995.
42. R. Giacobazzi, C. Palamidessi, and F. Ranzato. Weak relative pseudo-complements of closure operators. *Algebra Universalis*, 36(3):405–412, 1996.
43. V. Gupta, R. Jagadeesan, and V. Saraswat. Computing with continuous change. *Science of Computer Programming*, 30(1–2):3–49, Jan. 1998.

44. L. Henkin, J. Monk, and A. Tarski. *Cylindric Algebras (Part I)*. North-Holland, 1971.
45. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proc,. of ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM, New York, 1987.
46. M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Proc. of the Fourth International Conference on Logic Programming*, Series in Logic Programming, pages 858–876, Melbourne, 1987. The MIT Press.
47. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1980.
48. M. Nielsen, C. Palamidessi, and F. Valencia. On the expressive power of concurrent constraint programming languages. In *Proc. of PPDP'02*, pages 156–167. ACM Press, 2002.
49. M. Nielsen, C. Palamidessi, and F. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Computing*, 9(2):145–188, 2002.
50. C. Rueda, G. Alvarez, L. Quesada, G. Tamura, F. Valencia, J. Diaz, and G. Assayag. Integrating constraints and concurrent objects in musical applications: A calculus and its visual language. *Constraints*, 6(1), 2001.
51. V. Saraswat. A somewhat logical formulation of CLP synchronization primitives. In R. A. Kowalski and K. A. Bowen, editors, *Proc. of the Fifth International Conference on Logic Programming*, Series in Logic Programming, pages 1298–1314, Seattle, USA, 1988. The MIT Press.
52. V. Saraswat. *Concurrent Constraint Programming*. PhD thesis, Carnegie-Mellon University, January 1989. In ACM distinguished dissertation series. The MIT Press, 1993.
53. V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *LICS*, pages 71–80. IEEE Computer Society, 1994.
54. V. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. *J. Symb. Comput.*, 22(5/6):475–520, 1996.
55. V. Saraswat, R. Jagadeesan, and V. Gupta. jcc: Integrating timed default concurrent constraint programming into java. In *Proc. of EPIA*, 2003.
56. V. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of the seventeenth ACM Symposium on Principles of Programming Languages*, pages 232–245. ACM, New York, 1990.
57. V. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of Concurrent Constraint Programming. In *Proc. of the eighteenth ACM Symposium on Principles of Programming Languages*. ACM, New York, 1991.
58. E. Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, Institute for New Generation Computer Technology (ICOT), Tokyo, 1983.
59. E. Shapiro. Concurrent Prolog: A progress report. *Computer*, 19(8):44–58, 1986.
60. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *ICLP*, pages 127–138, 1984.
61. K. Ueda. Guarded Horn Clauses. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, Series in Logic Programming. The MIT Press, 1987.
62. K. Ueda. Guarded Horn Clauses, a parallel logic programming language with the concept of a guard. In M. Nivat and K. Fuchi, editors, *Programming of Future Generation Computers*, pages 441–456. North Holland, Amsterdam, 1988.
63. K. Ueda and K. Furukawa. Transformation rules for ghc programs. In *FGCS*, pages 582–591, 1988.

64. K. Ueda, N. Kato, K. Hara, and K. Mizuno. LMNtal as a unifying declarative language: Live demonstration. In *Proc. of ICLP 06*. LNCS, 2006.
65. E. Zaffanella. Domain Independent Ask Approximation in CCP. In U. Montanari and F. Rossi, editors, *Proc. First Int'l Conf. on Principles and Practice of Constraint Programming (CP'95)*, volume 976 of *Lecture Notes in Computer Science*, pages 362–379. Springer-Verlag, Berlin, 1995.
66. E. Zaffanella, R. Giacobazzi, and G. Levi. Abstracting synchronization in concurrent constraint programming. *Journal of Functional and Logic Programming*, 1997(6), November 1997.